Unit 1: Fundamental of Programming in Java

CEUE203 : Object Oriented Programming

By Dr. Parth Goel

Chapter Overview

- Introduces Java as a high-level, object-oriented, platform-independent programming language.
- Explains the role of the **Java Virtual Machine (JVM)** and the concept of **bytecode**.
- Discusses Java's **design philosophy** through eight core **buzzwords** like Simple, Robust, and Multithreaded.
- Covers **modern Java features** such as records, switch expressions, var, text blocks, and sealed classes.

Learning Objectives

- Explain the Java compilation process from .java source to .class bytecode.
- Describe how the **JVM** ensures platform independence and performance.
- Identify and apply Java's **core buzzwords** with practical examples.
- Use **new Java features** like switch expressions, records, and sealed classes in small programs.
- Understand how Java is used in **real-world applications** such as Android, Banking, and IoT systems.

History of Java's Origin

- Java was created in the early 1990s by **James Gosling** and his team at **Sun Microsystems**.
- Originally intended for **embedded systems** like TV set-top boxes.
- The internal project was called the **Green Project**.
- The first name of the language was **Oak**, named after an oak tree outside Gosling's office.

Why the Name “Java”?

- “Oak” was already a registered trademark, so a new name was needed.
- The developers picked **Java**, inspired by their love for Java coffee from Indonesia.
- The name symbolizes **energy**, **creativity**, and **simplicity** — values that reflected the language's design.

Design Goals of Java

- Designed to be **simple**, especially for developers with a C/C++ background.
- Focused on **portability**: code should run on **any platform** without changes.
- Emphasized **security** in the age of connected, networked systems.
- Needed to be **faster** than traditional interpreters through optimized execution methods.

Evolution of Java Versions

- **Java 1.0 (1995):** First public version, introducing applets and core libraries.
- **Java 2 (1998):** Split into Standard (J2SE), Enterprise (J2EE), and Micro (J2ME) editions.
- **Java 5.0 (2004):** Introduced powerful features like **generics**, **annotations**, and **enums**.
- **Java 8 to 21 (2014–present):** Added lambdas, streams, modules, and modern features like records, var, and sealed classes. Java now updates every 6 months.

Real-World Significance

- Java's **platform independence** made it the go-to for **Android app development**.
- Used in **enterprise-grade applications** like **banking**, **ERP systems**, and **government portals** due to its stability.
- Java's roots in embedded systems make it suitable for **IoT and smart devices**.
- Its continuous evolution makes it relevant for **modern cloud and web applications**.

Syntax Example Code

```java
public class WelcomeBank {
    public static void main(String[] args) {
        System.out.println("Welcome to the Java-powered Bank Management
System.");
    }
}
```

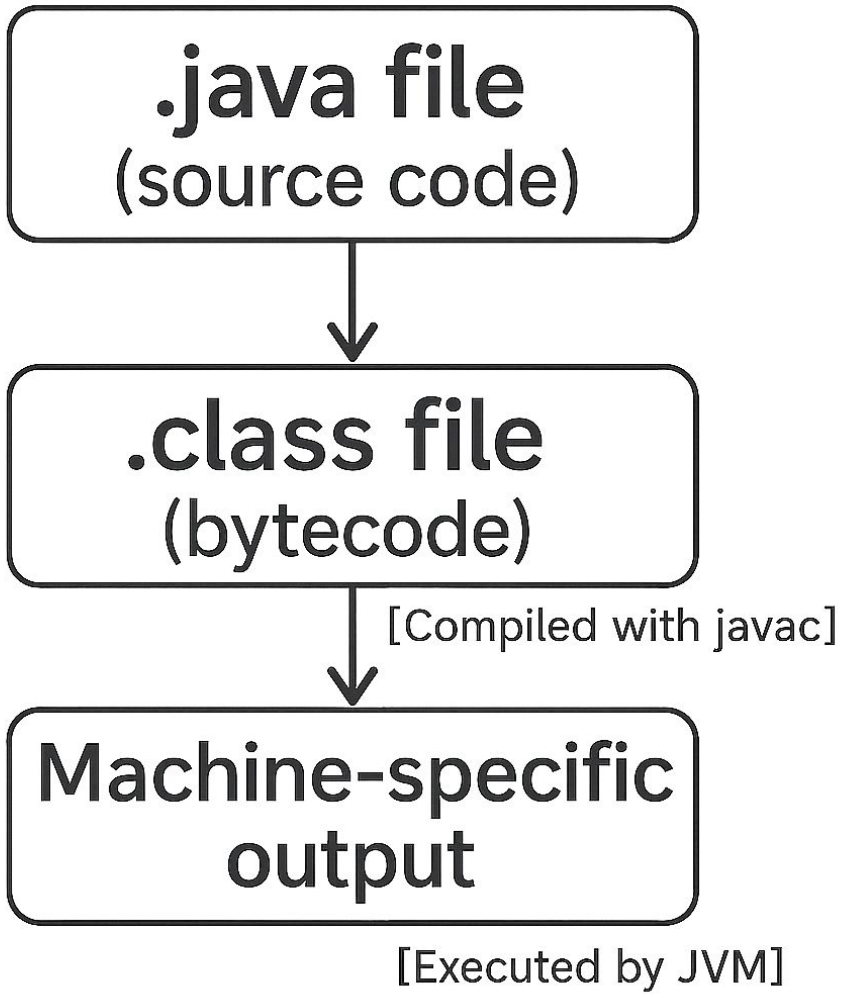
- This simple program prints a welcome message.
- The **same code** runs on **Windows, Linux, and macOS** without modification.
- Unlike C++, no need to recompile for each OS — Java uses **JVM** for this magic.

Flow of Compilation in Java

- Step 1: Write source code in a .java file.
- Step 2: Compile it using javac, which generates a .class file (Bytecode).
- Step 3: Run the .class file using the java command — this uses the **JVM** to execute the bytecode.
- **Separation of compilation and execution** is key to Java's portability.

Flow of Execution in Java

- **.java** is human-readable, like a script.
- **.class** is a cross-platform binary that the JVM understands.
- The **JVM interprets or compiles it** further into native instructions.



Why Java uses Bytecode?

- Bytecode allows the program to be **written once and run anywhere**.
- It doesn’t depend on the operating system like C/C++ binaries do.
- JVM adds a **security layer** by running code in a controlled environment.
- **Just-In-Time (JIT) Compilation** optimizes frequently used code for better speed.

Uses of the Bytecode

- A **banking application** can run on different OSs in regional branches using the same .class files.
- **Android applications** are written in Java and compiled to bytecode before conversion into Android-compatible formats.
- **Enterprise Java systems** (like web servers) use bytecode for platform-independent deployment.

Bytecode Example – Bank Customer Class

```java
public class BankCustomer {
    String customerName;
    double balance;

    public BankCustomer(String name, double initialBalance) {
        customerName = name;
        balance = initialBalance;
    }

    public void showDetails() {
        System.out.println("Customer Name: " + customerName);
        System.out.println("Balance: ₹" + balance);
    }

    public static void main(String[] args) {
        BankCustomer c1 = new BankCustomer("Name1", 15000);
        c1.showDetails();
    }
```

Compile:

javac BankCustomer.java
→ Generates BankCustomer.class

Run:

java BankCustomer
→ Executes via JVM

Understanding Java Compilation

- **Java source files can't run directly.**
  They must be compiled with javac to produce .class bytecode first.

- **Bytecode isn’t machine code.**
  It’s an intermediate format executed by the JVM on any OS.

- **Changes to .java need recompilation.**
  Otherwise, the program runs the old, outdated .class file.

- **Java isn’t slow anymore.**
  JVM uses JIT (Just-In-Time) compilation to optimize performance at runtime.

Introduction to Java Buzzwords

- Java’s design is built around 8 guiding principles called **buzzwords**.
- Each buzzword reflects a core strength of Java in real-world software.
- Understanding them helps in writing better, safer, and scalable Java code.

1st – Simple (Why Java is Simple)

- No pointers, reducing memory errors.
- Clean syntax similar to C, but more concise.
- No multiple inheritance through classes, avoiding ambiguity.
- Garbage collection manages memory automatically.

2nd – Object Oriented

(Everything is about Objects)

- All logic is written around **classes** and **objects**.
- Uses **encapsulation**, **inheritance**, **polymorphism**, and **abstraction**.
- Encourages modular, maintainable design through real-world modeling.

3rd – Robust (Built for Reliability)

- Strongly typed: All variables must have a declared type.
- Exception handling via try-catch ensures errors don’t crash the system.
- Automatic garbage collection prevents memory leaks.
- No pointer arithmetic reduces security risks.

4^{th} – Multithreaded

(Concurrency Made Easy)

- Java supports **parallel execution** using **threads**.
- You can run multiple tasks simultaneously within a program.
- Helps in building responsive apps, e.g., handling user input while fetching data.
- Thread safety is achieved using the synchronized keyword.

5th – Architecture Neutral

("Write Once, Run Anywhere")

- Java compiles code into **bytecode**, not machine-specific instructions.
- Bytecode runs on any platform using the **JVM**.
- Same .class file works on Windows, Mac, or Linux without changes.

6^{th} – Interpreted & High Performance

(Fast and Flexible)

- JVM interprets bytecode line-by-line or compiles it on the fly.
- **JIT Compiler** converts hot code paths to native code during execution.
- This brings Java's performance close to compiled languages like C++.

7th – Distributed (Built for Networks)

- Java includes libraries for **network programming** (e.g., java.net, RMI).
- Ideal for client-server systems, cloud apps, and IoT.
- Supports communication between multiple machines securely and efficiently.

8th – Dynamic (Adaptable at Runtime)

- Java can load classes and objects during execution.
- Supports **reflection** to inspect or modify code on the fly.
- Useful in IDEs, plugin systems, and dynamic web applications.

Summary of Buzzwords

- **Simple** – Easy to write and read.
- **Object-Oriented** – Everything is a class or object.
- **Robust** – Handles errors and memory well.
- **Multithreaded** – Runs many tasks at once.
- **Architecture Neutral** – Same code on any OS.
- **Interpreted & High Performance** – Flexible yet fast.
- **Distributed** – Ready for networked systems.
- **Dynamic** – Evolves during runtime.

Buzzwords in Action

- **Android apps** use multithreading, OOP, and exception handling.
- **Banking systems** rely on robustness, architecture neutrality, and security.
- **Web services** benefit from distributed and dynamic behavior in microservices.

Buzzwords in Banking System

- **Object-Oriented**:
  We model the bank account using a **class** with fields and methods.

- **Robust**:
  We prevent crashes and wrong deductions using **conditional logic** and type safety.

- **Multithreaded**:
  Two customers can withdraw **at the same time** using **threads**.

- **Simple**:
  The code is easy to write and understand, even for beginners.

- **Safe Access (Synchronization)**:
  Shared data is protected using the synchronized keyword to **prevent race conditions**.

Modern Java and its Evolution

Java was once considered **verbose and boilerplate-heavy**. Newer languages (like Kotlin, Python) forced Java to evolve. From Java 10 to Java 21, features were added to **make Java cleaner, expressive, and developer-friendly**, such as:

- Switch expressions
- Records
- Text blocks
- Pattern matching
- var
- Sealed classes

Switch Expressions

- Traditional switch was clunky, required break, and error-prone.
- Java 14 introduced **switch expressions** with → syntax.

Why it Matters

- Reduces code clutter.
- Prevents fall-through bugs.
- Can **return values** directly, making switch usable in expressions.

Switch Expressions Example

```java
public static String getAccountDescription(String type) {
    return switch (type) {
        case "SAVINGS"   → "Savings Account with interest";
        case "CURRENT"   → "No interest, flexible transactions";
        case "FIXED"     → "Locked deposit with higher interest";
        default          → "Invalid account type";
    };
}
```

- Each case returns a value directly.
- No need for break.
- default still required to catch invalid inputs.

Old v/s New Switch

Old switch Problems

- Needs manual break after each case.
- Allows accidental fall-through if break is missed.
- Cannot directly return a value.

New switch Fixes

- Uses → instead of : for clarity.
- Prevents fall-through.
- Can be assigned to a variable (like an expression).
- Cleaner syntax for grouped cases.

Common Errors with Switch

- Using : instead of → in modern switch — syntax error.
- Forgetting default case — leads to unhandled inputs.
- Trying to use in older Java versions — only available from **Java 14+**.
- Using multi-line logic without {} and yield — must wrap and return value properly.

Records – Simpler Containers

- Data classes used to need **tons of boilerplate**: constructors, getters, toString(), equals()...

- Records give you all that **in one line**.

Using a Record

```java
record Customer(int accountNumber, String name, String accountType) {}

public class BankCustomerDemo {
    public static void main(String[] args) {
        Customer cust = new Customer(101, "Name1", "SAVINGS");
        System.out.println(cust); // Prints readable string
    }
}
```

- Fields are **immutable**.
- Auto-generates constructor, toString(), equals(), and hash code.
- Accessor methods use field names: cust.name() instead of getName().

Records v/s Classes

Use Records When:

- Needing a class that only **holds data**, not behavior.
- Desiring **immutability** and less code.

Use Classes When:

- Needing **mutable fields**.
- Planning to **extend or inherit behavior**.
- Requiring to override methods manually or add extra logic.

Common Errors with Records

- Trying to change field values — **records are immutable**.
- Using getField() — record accessors are **just the field name**.
- Trying to subclass a record — records are **implicitly final**.
- Compiling on Java version < 16 — not supported.

Text Blocks in Java

```
String html = """
    <html>
        <body>
            <h1>Welcome</h1>
        </body>
    </html>
""";
// no need to escape characters
// or break lines manually
```

- Writing long strings like SQL, HTML, or JSON was messy with "\n" and +.
- **Text blocks**, introduced in **Java 15**, let you write multi-line strings easily using """.

Text Block Example

```java
public class AccountSummary {
    public static void main(String[] args) {
        String summary = """
            Account Summary:
            ------------------
            Name: Name1
            Type: Savings
            Balance: ₹25,000
            """;
        System.out.println(summary);
    }
}
```

- **Output is formatted exactly as written**, with correct indentation and line breaks.
- Useful for printing reports, messages, templates, or logs.

Pattern Matching with instanceof

```java
if (obj instanceof Customer) {
    Customer c = (Customer) obj;
    System.out.println(c.name());
}
// Old Way
```

```java
if (obj instanceof Customer c) {
    System.out.println(c.name());
}
// New Way
```

- **Simplifies type checking and casting.**
- Reduces repetitive casting syntax.
- More readable and safer.

var Keyword in Java

- Introduced in **Java 10**, var lets the compiler **infer the variable type**.
- var makes local code **cleaner**, but type is still fixed at **compile-time**.

var Keyword in Java

Use var When:

- The **type is obvious** from the right-hand side.

Example: var count = 100; → Clearly an int.

- Wanting to **reduce clutter**, especially with long generic types.

Example: var map = new HashMap<String, List<Integer>>();

- The focus is on the **logic**, not the data type.

Example: inside loops or short method blocks.

Avoid var When:

- The expression's type is **not obvious**.

Example: var data = process(input); — what's data?

- It makes the code **less readable**, especially for others reviewing it.

- Working with APIs or types that are **critical to know explicitly**.

For example, don't hide return types like File, Socket, Thread, etc.

Sealed Classes

- Introduced in **Java 17**, sealed classes allow you to **limit which classes or interfaces can extend or implement a type**.
- It's a way to declare: “Only these specific subclasses are allowed.”
- Improve **security** by preventing unknown or unauthorized subclassing.
- Enforce **closed inheritance** where you want controlled polymorphism.
- Makes code more **predictable**, especially in large team projects.

Sealed Classes

- A sealed class must use the permits clause to **explicitly list allowed subclasses**.

- Subclasses must be marked as either:
  - final – no further subclassing allowed,
  - sealed – continue sealed chain, or
  - non-sealed – open inheritance.

Example for Sealed Class

```java
// Sealed Superclass
public sealed class Account permits SavingsAccount, LoanAccount {
    public void displayType() {
        System.out.println("General Account");
    }
}

// Permitted Subclasses
final class SavingsAccount extends Account {
    public void displayType() {
        System.out.println("Savings Account");
    }
}

final class LoanAccount extends Account {
    public void displayType() {
        System.out.println("Loan Account");
    }
}
```

- Account can **only be extended** by SavingsAccount and LoanAccount.
- No one can create another subclass like CreditCardAccount unless explicitly permitted.
- Enhances **code safety**, especially in **financial systems** where business logic must stay tightly controlled.

Chapter Summary

- Java’s **history and philosophy** — simple, portable, secure, high-performance.
- The **compilation model** — from .java source to .class bytecode via JVM.
- Understanding Java's **buzzwords** — from "Simple" to "Dynamic".
- Use of **multithreading**, **exception handling**, and **OOP** principles.
- Modern Java features like:
  - switch expressions for concise control flow,
  - records for compact data classes,
  - text blocks for readable multi-line strings,
  - pattern matching with instanceof,
  - var for cleaner variable declarations,
  - and sealed classes for secure inheritance.

Thank You